NASA Technical Memorandum 106597

Implementing Software Safety in the NASA Environment

Martha S. Wetherholt
Lewis Research Center
Cleveland, Ohio

and

Charles F. Radley
EBASCO Services, Inc.
Brook Park, Ohio

Prepared for the
Safety Through Quality Conference
sponsored by the Real Time Associates, Ltd.
Old Windsor, Berkshire, United Kingdom, June 6–7, 1994

National Aeronautics and
Space Administration

Implementing Software Safety in the NASA Environment

Martha S. Wetherholt
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Charles F. Radley
EBASCO Services, Inc.
Brook Park, Ohio 44142

ABSTRACT. Until recently, NASA did not consider allowing computers total control of flight systems. Human operators, via hardware, have constituted the ultimate safety control. In an attempt to reduce costs, NASA has come to rely more and more heavily on computers and software to control space missions. (For example, software is now planned to control most of the operational functions of the International Space Station.) Thus the need for systematic software safety programs has become crucial for mission success.

Concurrent engineering principles dictate that safety should be designed into software up front, not tested into the software after the fact. 'Cost of Quality' studies have statistics and metrics to prove the value of building quality and safety into the development cycle. Unfortunately, most software engineers are not familiar with designing for safety, and most safety engineers are not software experts. Software written to specifications which have not been safety analyzed is a major source of computer related accidents.

Safer software is achieved step by step throughout the system and software lifecycle. It is a process that includes requirements definition, hazard analyses, formal software inspections, safety analyses, testing, and maintenance. The greatest emphasis is placed on clearly and completely defining system and software requirements, including safety and reliability requirements. Unfortunately, development and review of requirements are the weakest link in the process. While some of the more academic methods, e.g. mathematical models, may help bring about safer software, this paper proposes the use of currently approved software methodologies, and sound software and assurance practices to show how, to a large degree, safety can be designed into software from the start.

NASA's approach today is to first conduct a preliminary system hazard analysis (PHA) during the concept and planning phase of a project. This determines the overall hazard potential of the system to be built. Shortly thereafter, as the system requirements are being defined, the second iteration of hazard analyses takes place, the systems hazard analysis (SHA). During the systems requirements phase, decisions are made as to what functions of the system will be the responsibility of software. This is the most critical time to affect the safety of the software. From this point, software safety analyses as well as software engineering practices are the main focus for assuring safe

software. While many of the steps proposed in this paper seem like just sound engineering practices, they are the best technical and most cost effective means to assure safe software within a safe system.

THE CURRENT NASA ENVIRONMENT

NASA, for the most part, is a research and development organization where development of hardware - rockets, jet engines, turbo props, satellites, etc. - has been, and still is, the primary focus. Software, coming late into this hardware dominated environment, has typically been viewed as; 1) highly suspect, 2) a low cost catch-all, and 3) necessary but not worth putting too much effort into. In the cases where it is viewed as highly suspect, there have always been hardware back-ups and crew operations to work around the software. Ten to fifteen years ago, in the few situations where software had to be used for some safety critical operation, N-version programming with voting logic was used (e.g. shuttle General Purpose Computers). However, that approach is too costly for most programs today.

Hardware is more familiar. The ways to develop, test and operate hardware have a long standing tradition and are trusted. Software, on the other hand, is not generally well understood by managers, the majority of which are hardware oriented. It makes sense and can be considered sound engineering to rely on what is most familiar, tried and true - the hardware and the ground and flight crew which operate it. Thus, software was relegated to non-safety related tasks with only a few exceptions.

Software's greatest asset is often, paradoxically, its greatest liability. Software is flexible. It can perform a multitude of tasks with little power or space. Software controlled hardware is often lighter and more standardized. Its very versatility, its ability to change without retooling or new weight calculations, its ability to provide information quickly and in human readable formats with less space and lower power consumption, are the very reasons why we must exploit all the possibilities software presents us. However, being considered "so easy to change" often leads to many of the problems encountered with software. Software designs are either left until last in a project or, if started early, keep changing throughout the project because there is very little understanding of the actual implications of how software is properly designed and built. On one hand, software is not trusted, on the other hand, the poor processes used in developing software lead to justification of that mistrust.

NASA has, over the course of its many programs and projects, tended to let each center, even each project within a center, choose it's own software standards. Unlike our military, which is strict about having one set of standards and enforcing contractor compliance to

them, NASA (excluding hardware safety) has only 'guidelines' and 'suggested' standards. That is not to say that there are no standards applied. It is merely up to each project manager to choose the standards he/she thinks are best suited for their project. There are no 'standard', or required, standards except for hardware safety standards [1] and most of those are space shuttle payload specific. This becomes a problem when large dissimilar systems have to interface or interoperate. The largest NASA systems, e.g. Space Shuttle and Kennedy Space Center launch control systems, were developed in isolation without regard to interoperability and with contractors using their own standards.

NASA is in the process of changing its approach to developing projects that utilize software and, hopefully, this will lead to a change in attitude about software's role as well. More and more sophisticated controls and monitoring are needed as we do more and more work in space and on the ground. Software is performing safety critical tasks, despite claims to the contrary. So, like hardware, software must also rely on a set of safety standards that must be followed.

It is difficult for NASA, as a whole, to accept the need for software standardization for a variety of reasons. To some degree, a distaste for rules and regulations that is traditionally found in many software engineers as well as a lack of understanding of software in R&D oriented hardware managers, has lead to questioning the need to place so much emphasis on a comprehensive software process. Also, many managers and engineers are accustomed to working on small projects. In this environment, different management skills are stressed; the same five to ten people on a project have usually conceived, designed, built, coded, and tested all the hardware and software. Thus, due partly to the lack of opportunity, there is a need to build the broader management skills and appreciation for high level planning needed for today's more complex systems. On larger systems, contractors handle the majority of software management and choose the standards, usually with limited guidance from NASA and a dictate that software "shall not be used in critical applications". While there are many successes, projects like Space Station have drawn attention to NASA's difficulties. Both the Aeronautical Safety Advisory Panel (ASAP) and the National Research Council (NRC) [2] have pointed out to NASA the need for software standardization, especially where safety is concerned.

THE PATH TO CHANGE

This paper presents the emerging NASA approach to software safety. First, NASA's goal must be to have effective software standards and techniques that are adhered to by all developers of NASA software, both in-house and by our contractors. Second, a comprehensive set of software safety specific requirements and analyses must be

defined, understood, and used.

NASA is still working to achieve the first goal. As the Aerospace Safety Advisory Panel annual report of March, 1994 shows:

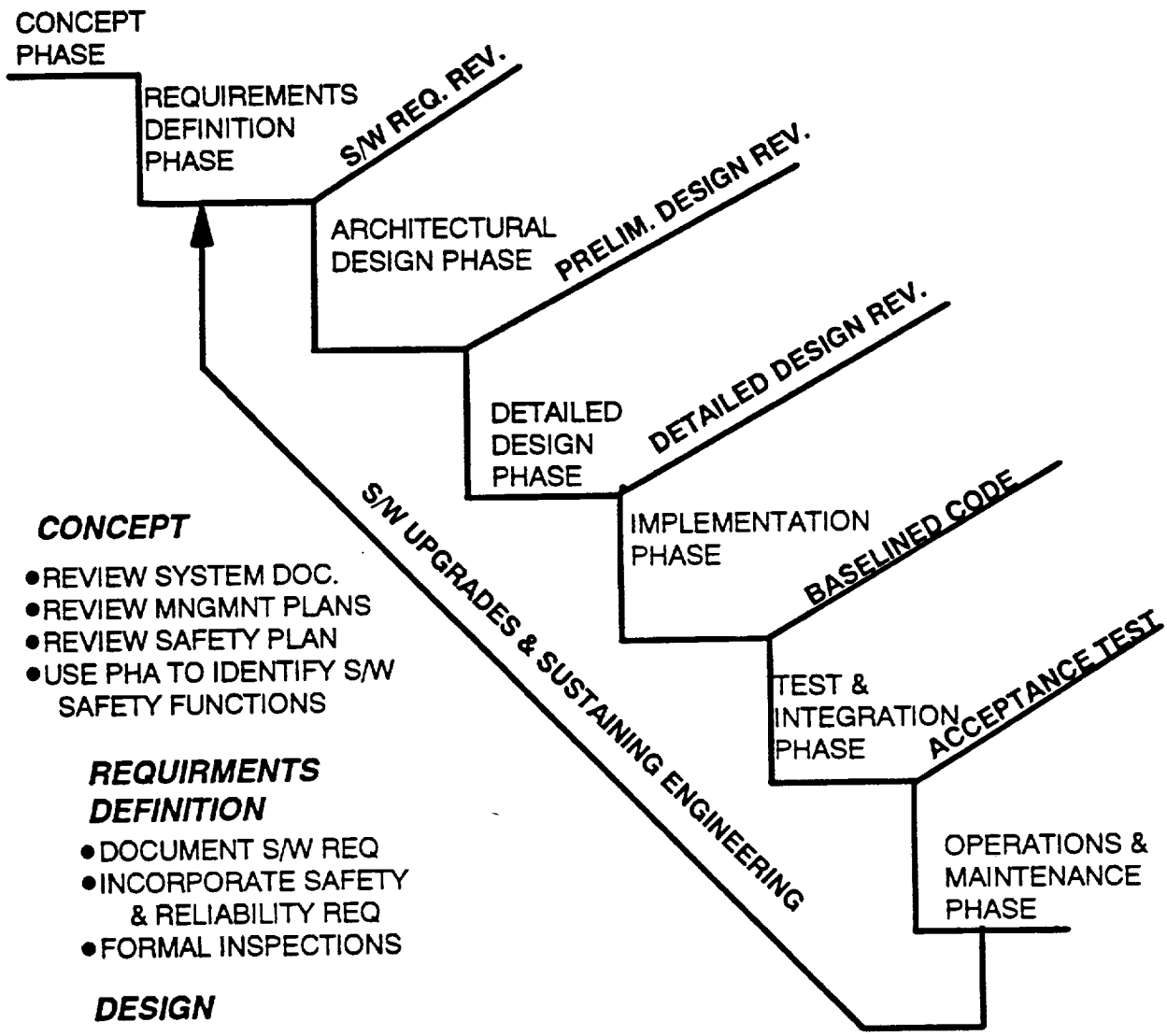
> "**Finding #31:** NASA's past approach to software development has been to incorporate it within the individual programs, allowing them to determine their own requirements and development, verification and validation procedures. In the future, as the complexity of NASA's computer systems and the need for interoperability grow, this mode of operation will be increasingly less satisfactory. While NASA has some good software practices, it does not have the overall management policies, procedures, or organizational structure to deal with these complex software issues."
>
> "**Recommendation #31:** NASA should proceed to develop and implement an agency wide policy and process for software development, verification, and safety as quickly as possible."

A few common techniques, procedures and practices are starting to be applied within, if not across, the NASA centers. Some of these practices are briefly discussed.

A Software Safety Standard (SSS) which addresses both the software safety requirements and the safety analyses, is now in review across NASA. It is due to be published by fall, 1994. Like its hardware predecessor, it must be universally applied. The basic approach to software safety as described in the standard, is outlined below. A NASA Software Safety Guidebook will provide more of the explanations of how, where, and where not, to apply the software safety requirements and analyses put forth in the Software Safety Standard. The guidebook will be completed about a year after the standard is released.

THE SOFTWARE DEVELOPMENT LIFECYCLE

NASA usually views the software lifecycle as a waterfall. While proto-typing and spiral lifecycle methods are used, they are not currently the typical approach. Thus most of our terminology is based on the phases of the waterfall lifecycle model. While not explicitly discussed in the Software Safety Standard, most of the software safety process is applicable to all lifecycle models. Figure 1 shows the typical software waterfall lifecycle, its milestones, and the usual software tasks performed in each progressive step (or phase). Figure 2 shows the software safety lifecycle.



CONCEPT

- REVIEW SYSTEM DOC.
- REVIEW MNGMNT PLANS
- REVIEW SAFETY PLAN
- USE PHA TO IDENTIFY S/W SAFETY FUNCTIONS

REQUIRMENTS DEFINITION

- DOCUMENT S/W REQ
- INCORPORATE SAFETY & RELIABILITY REQ
- FORMAL INSPECTIONS

DESIGN

- DESIGN & DOCUMENT
- INCORPORATE S/W SAFETY FEATURES
- FORMAL INSPECTIONS
- S/W FAULT ANALYSIS
- FDIR DESIGN
- TEST PLANS

IMPLEMENTATION

- CODING
- UNIT TEST
- FDIR
- SAFETY FEATURES
- FORMAL INSPECTIONS

TESTING

- CSCI TESTING
- S/W INTEGRATION TESTS
- H/W INTEGRATION TESTS
- ACCEPTANCE TESTS
- DATA ACCEPTANCE PKG

Figure 1. The Software Waterfall Lifecycle

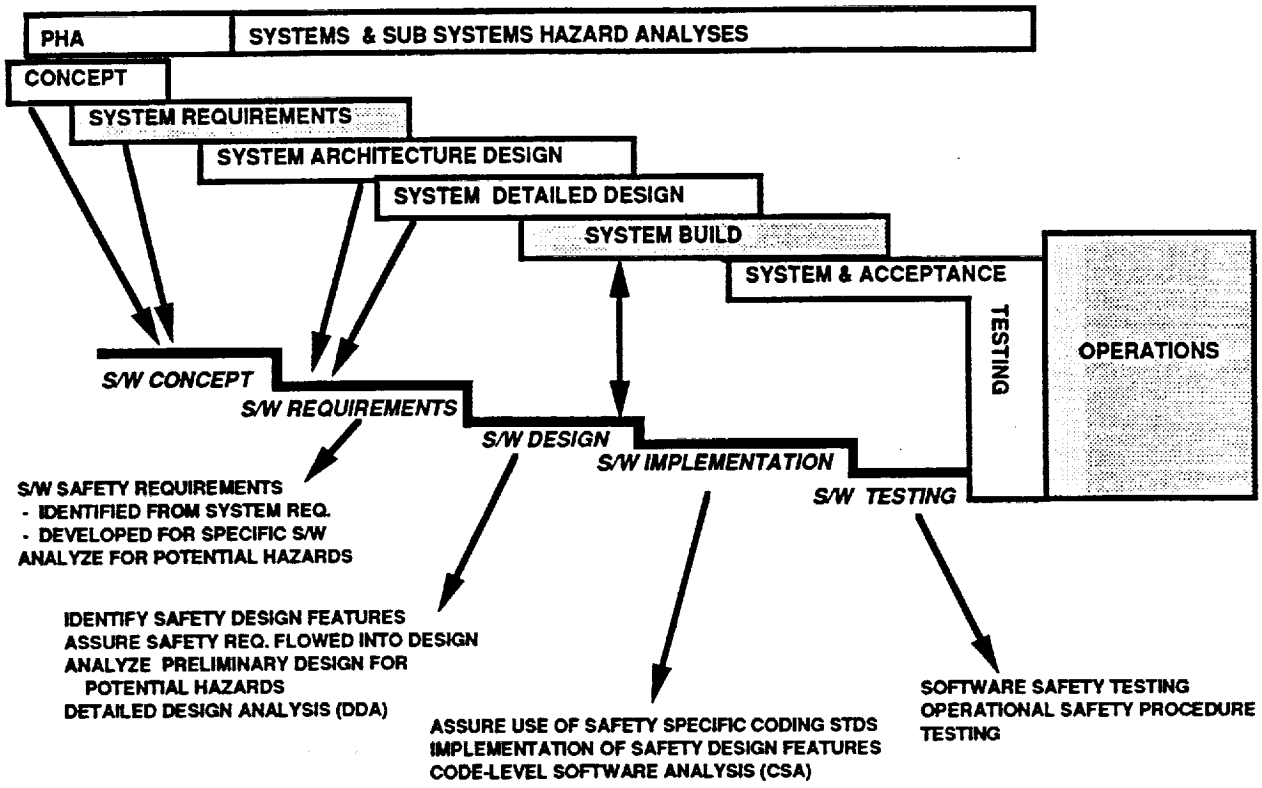


Figure 2. System & Software Lifecycles and the Safety Tasks

JUST PLAIN GOOD DEVELOPMENT PRACTICES

The overall software development process is very important to achieving complete, safe and reliable software. What good are safety requirements and hazard report recommendations if the software or the software development process is the major cause of errors? Software development must have, as a minimum:

- software management plans that are followed;
- configuration management of all documents, code and tests;
- complete and unambiguous definition of all requirements early in the lifecycle;
- use of software language standards;
- use of Yourdon, de Marco, etc. design techniques;
- clear, well defined interfaces;
- documented code that helps those that must maintain it;
- comprehensive tests plans and procedures for unit through acceptance testing;
- problem tracking system that ties into the system problem tracking during the software and hardware integration tests;
- a software product assurance program that, as a minimum,

assures the software completely meets, tests, and delivers all requirements, the documentation is complete and up to date, and established standards have been followed, or appropriately waived;
- pre-flight reviews that incorporate software status;
- sign-off/acceptance procedures which include data acceptance packages to insure what is shipped is what is meant to fly.

NASA has standards and/or guidelines which address all of the above mentioned practices. However, many developers/project managers at NASA may use a modified/tailored DoD-STD-2167A, IEEE software standards, or other approaches.

There is a lot of work that must go into achieving complete, safe and reliable software. However, schedules are tight and budgets even tighter. We need to get the most "bang for our buck", that is put the most effort into what will yield the highest pay-off. Studies indicate that the most costly software errors ("error - Human action that results in software containing a fault." IEEE definition) occur during the requirements definition phase of the software lifecycle.

Robyn Lutz of the Jet Propulsion Laboratory (JPL), in a study of errors found during integration and systems testing of the Voyager and Galileo spacecraft [3] revealed that the majority of safety related errors resulted from failure to identify or understand functional requirements (62% on Voyager, 79% on Galileo). Of those missing functional requirements, the majority were from misunderstood hardware and software interface specifications (65% Voyager, 48% Galileo). Non-safety related functional faults were usually caused during implementation of the requirements.

As an error propagates through the software development lifecycle, it becomes more and more expensive to fix as each requirement becomes documented, designed to, implemented, and tested prior to being found incorrect or missing. The 'cost of quality' is very evident in estimating the cost to correct software errors. To fix a problem when it is created costs little. Finding the problem in the next development phase costs 10 times what it would have cost to fix the problem before. If the problem is found during the operational phase, it is 100 times more expensive to fix. Thus, NASA is concentrating on getting the requirements correct first. The next level of emphasis is to ensure that most errors are found during the phase in which they are inserted. The old idea of testing safety and quality into a product have just not proved to be technically or cost effective. Software is notoriously difficult, even impossible, to test for all possible paths that can be taken.

Software Formal Inspections [4] is one method for finding errors at the point within the development cycle in which they occur. It has built in checklists that target each software lifecycle phase.

SFI has proved effective in discovering missing, incomplete and wrong requirements early in the development process. JPL has developed training, standards and guidelines for use of the Formal Inspection process. The process was based on methods originally created in 1972 by Michael Fagan when he was at IBM [5,6] and expanded by John Kelly of JPL [7]. Software Formal Inspections (SFI) are a series of very structured, intra-lifecycle phase, peer reviews of the non-released software product (documents, code, test procedures, etc.). The method is straight forward, well laid out and provides built in metrics. The results are quite impressive, using formal inspections, the hours needed to fix a defect found by SFI is about .7 hours as opposed to 5-18 hours if the defect is left to be found during testing.

This method is being adopted, albeit slowly, within most of the NASA centers. In addition to its built in metrics, checklists and straight forward process, Formal Inspections target errors within the lifecycle in which they are created. Taking small pieces of the product and reviewing it in a non-adversarial atmosphere with only those peers and experts which can contribute technically (i.e. without management involvement), builds the team work and empowerment aspects so important in NASA's goal to embrace Total Quality Management. To a large extent, besides its logical methodology, the growing success of Formal Inspections can be attributed to the training and support that JPL provides to each center to get the center started. NASA headquarters has further supported this methodology by issuing a NASA Software Formal Inspection Standard and Guidebook which follow the JPL SFI training course.

THE NASA SOFTWARE SAFETY STANDARD

While a separate NASA Software Safety Standard is being created, it is heavily stressed that software safety is a part of the overall system safety effort and is not performed in isolation to the rest of the system. Software gets its first indication of criticality from the preliminary system safety analysis. As the iterative process of both the system and software safety analysis progresses to deeper levels of detail, information from each analysis flows to the other.

Due to some uniqueness of software safety analyses and in order to call attention to the need for a software safety program/methodology within NASA, the Software Safety Standard is currently a separate document and not combined within the NASA Safety Policy and Requirements Document. This Software safety standard is to be applied to all software acquired by NASA and all software developed by NASA that is used as a part of a system that possesses the potential of directly, or indirectly, causing harm to humans or damage to property exterior to the system. When

software is acquired by NASA, this standard is to be specified in contract clauses or memoranda of understanding. When software is developed by NASA, this standard applies and will be specified in the program plan, software management plan, or other controlling document.

Software safety requires a coordinated effort among all organizations involved in the development of NASA software. Those conducting the Software Safety effort must interface with personnel from disciplines such as reliability, Independent Verification and Validation (when available), and human factors.

The purpose of the software safety process is to ensure that software does not cause, or contribute to, a system reaching a hazardous state; that it does not fail to detect or take corrective action if the system reaches a hazardous state; and that it does not fail to mitigate damage if an accident occurs.

The overall software safety process is to:

a. Ensure that the system/subsystem safety analyses identify which software is safety-critical. Any software that has the potential to cause a hazard or is required to support control of a hazard, as identified by safety analyses, is safety-critical software.

b. Ensure that the system/subsystem safety analyses clearly identify the key inputs into the software requirements specification e.g., identification of hazardous commands, limits, interrelationship of limits, sequence of events, timing constraints, voting logic, failure tolerance, etc.

c. Ensure that the development of the software requirements specification includes the software safety requirements that have been identified by software safety analysis.

d. Ensure that the software design and implementation properly incorporates the software safety requirements.

e. Ensure that the appropriate verification and validation requirements are established to ensure proper implementation of the software safety requirements.

f. Ensure that test plans and procedures satisfy the intent of the verification requirements.

g. Ensure that the results of the verification effort are satisfactory.

Software safety procedures are to be performed as an integrated

activity of the system safety effort. Those performing software safety provide support during system concept definition, safety planning, design of system architecture to minimize safety critical configuration items, and identification of Safety Critical Computer Software Components (SCCSCs). Using the results of the system Preliminary Hazard Analysis (PHA) and system safety requirements allocated to software as the starting point for SCCSC identification, software safety continues to work software safety requirements definition and analysis activities throughout software development and test.

The software safety effort provides hazard analysis reports, software safety analyses and testing results to system safety, on a continuous basis, for inclusion in the System Hazard Analysis and Integrated Hazard Analysis.

RISK ASSESSMENT

Once identified, a potential hazard may be 1) eliminated, 2) mitigated, or 3) accepted. Hazards are categorized based on both severity and likelihood of occurrence with "1" being the most critical. (See Figure 3.) The order of precedence for satisfying system and software safety requirements and for resolving identified hazards is as follows:

a. Design for Minimum Risk. From the onset, design to eliminate hazards. If an identified hazard cannot be eliminated, reduce the associated risk to an acceptable level, as defined by management, through design selection.

b. Incorporate Safety Devices and/or Failure Tolerance (These can be either software, hardware, or software and hardware combined). If identified hazards cannot be eliminated or their associated risk adequately reduced through design selection, that risk is to be reduced to a level acceptable to the management through the use of fixed, automatic, or other protective safety design features or devices, or redundancy.

c. Provide Warning Devices (These can be either software, hardware, or software and hardware combined). When neither design nor safety features/devices can effectively eliminate identified hazards or adequately reduce associated risk, devices are to be used to detect the condition and to produce an adequate warning signal to alert personnel of the hazard. Warning signals and their application are to be designed to minimize the probability of incorrect personnel reaction to the signals and be standardized within like types of systems.

d. Develop Procedures and Training. Where it is impractical to eliminate hazards through design selection or adequately reduce the associated risk with safety and warning devices, procedures and training are to be used. However, without a specific waiver, no warning, caution, or other form of written advisory can be used as the only risk reduction method for Category 1 and 2 hazards (See Figure 3).

The decision (based on all relevant factors) to accept a hazard with its associated risk is a management responsibility, and requires coordination and concurrence by the designated safety official and the Program Manager. If there is a lack of concurrence on the decision between management and safety at any level, those performing safety on that project will elevate the decision to the next Safety, Reliability, Maintainability, and Quality Assurance (SRM&QA) management level. The probability of mishap coupled with the severity of the possible consequences is the major consideration in that decision. The Risk Assessment approach to determine the hazard category is shown in Figure 3.

		LIKELIHOOD OF OCCURRENCE			
		PROBABLE	**OCCASIONAL**	**REMOTE**	**IMPROBABLE**
SEVERITY LEVELS	**CATASTROPHIC**	1	1	2	3
	CRITICAL	1	2	3	4
	MARGINAL	2	3	4	5

Figure 3. Safety Categorization

SYSTEM SAFETY

The Software Safety process really begins during the system concept and requirements phase. The system Preliminary Hazard Analysis (PHA), and subsequent system and software safety analyses,

initially identify when software is a potential cause of a hazard or will be used to support the control of a hazard. This software is classified as safety-critical and is subjected to software safety analysis. Safety-critical software is typically 1) software which exercises direct command and control over potentially hazardous functions and or hardware, 2) software that monitors critical hardware components, 3) software which monitors the system for possible critical conditions and/or states, and/or 4) software that if not executed or is executed incorrectly, inadvertently, or out of sequence could result in a hazard or allow a hazardous condition to exist.

The system safety analyses are the first place to identify software safety requirements necessary to support the development of the software requirements specification. These requirements are then provided to the developer for inclusion into the software requirements document. Some examples of software safety requirements include limits (e.g., redlines, boundary values), sequence of events, timing constraints, interrelationship of limits, voting logic, hazardous hardware failure recognition, failure tolerance, caution and warning interfaces, hazardous commands, etc.

Within each phase of the software development lifecycle, two interrelated safety activities take place. One is the safety analysis of the deliverables produced within that lifecycle phase. These analyses determine if new potential hazards have arisen and if previously identified hazards have been properly removed or mitigated. The other activity consists of providing or building into the process and products known software safety requirements, practices, implementation techniques, and test methods.

SOFTWARE REQUIREMENTS PHASE

There are two main software safety tasks that are performed at this stage of the lifecycle, 1) further development of software safety requirements and 2) analysis of the software requirements for potential hazards.

Analysis of the software requirements can reveal potential hazards that the system safety analysis was unable to surface or can show where system requirements were not flowed into the software properly. These potential hazards can then be addressed by adding or changing system and/or software requirements.

Software Formal Inspections can have their highest affect when used at this stage of the software development lifecycle. The procedures help ensure that all appropriate personnel (e.g. system engineers, hardware designers, users, etc.) work together to focus on ensuring that all the requirements are in place and correctly defined.

SOFTWARE ARCHITECTURE/PRELIMINARY DESIGN PHASE

The software architectural design process develops the high level design that implements the software requirements. This includes all software safety requirements. During the design process, the software safety engineer identifies safety design features and methods (e.g., inhibits, traps, interlocks and assertions) that can be used throughout the software to implement the software safety requirements. Safety specific coding standards are also developed which identify requirements for annotation of safety-critical code and limitation on use of certain language features which can reduce software safety. After allocation of the software safety requirements to the software design, the next level SCCSCs are identified. These are all software components which implement software safety requirements or components which interface with SCCSCs which can affect their output.

Analysis is performed on the architectural design to identify potential hazards and on the test plans to verify incorporation of safety related testing. Input/ output timing, multiple event, out-of-sequence event, failure of event, wrong event, inappropriate magnitude, incorrect polarity, adverse environmental, deadlocking, and hardware failure sensitivities are included in the analysis.

SOFTWARE DETAILED DESIGN PHASE

After development of the detailed design, unit level SCCSCs are identified. These are all software units that implement software safety requirements or units which interface with SCCSCs which can affect their output.

During this phase, safety-related information is incorporated into all user manuals. This information includes cautions, warnings, and procedures for handling safety related procedures and hazards. These documents include the User's Guides and Operational Procedures. Software Formal Inspections of these documents will help assure the incorporation of the appropriate safety features.

Test cases which verify the software safety requirements and identify potential hazards are developed during this phase. The safety related test cases are to ensure that the software responds correctly to potential hazards and does not initiate any hazards. These test cases support Computer Software Component Item (CSCI), system and acceptance level testing.

A Detailed Design Analysis (DDA) is performed on the design to identify potential hazards and on test cases to ensure incorporation of safety related testing. A comprehensive Fault Detection, Isolation, and Recovery (FDIR) philosophy should be

incorporated in the design as well as implementation of the caution and warning requirements and further breakout of safety requirements. All of which are analyzed and related to the system safety requirements to assure that the software design is still on target with system requirements, design, and implementation.

SOFTWARE IMPLEMENTATION/CODING PHASE

The software implementation translates the detailed design into code in the selected programming language. As part of the process, the detailed design for the SCCSC is translated into the code in accordance with the safety specific coding standards. The code also implements any safety design features and methods developed during the design process.

Safety-critical code is to be commented with enough information and warning so that any future changes can be made with a reduced probability of introducing new software hazards.

Test procedures which support CSCI, system and acceptance level testing are developed during this phase to verify the software safety requirements and identify potential hazards. The safety related procedures include negative, no-go and stress testing to ensure that the software responds correctly to potential hazards and does not initiate any hazards.

The safety engineer performs and documents a Code-Level Software Analysis (CSA). Using the results of the Detailed Design Analysis, if previously accomplished, the safety activity analyzes program code and system interfaces for events, faults, and conditions that could cause or contribute to undesired events affecting safety. This analysis starts when coding begins and continues throughout the system life cycle. The results of the CSA are presented at software in-process reviews and system level safety reviews.

Activities to be accomplished during CSA are described below:

a. Analyze:

1) SCCSCs for correctness and completeness, and for input/output timing, multiple event, out-of-sequence event, failure of event, adverse environment, deadlocking, wrong event, inappropriate magnitude, improper polarity, and hardware failure sensitivities.

2) Software implementation of safety criteria called out in the system specifications and requirements documents.

3) Possible combinations of hardware failures,

software failures, transient errors, and other events that could cause the system to operate in a hazardous manner.

4) Proper error handling for inappropriate or incorrect data in the input data stream.

5) Fail-safe and fail-soft modes.

6) Input overload or out-of-bound conditions.

b. Perform an internal path and control process flow analysis on SCCSCs.

c. Any resulting design, coding, and testing change recommendations must be documented first in a hazard report or safety analysis report. Approved changes are then incorporated into the codes, design and if need be, the requirements specification and test documents as well.

d. Use formal methods and Software Formal Inspections procedures to verify that the as-written code conforms to the logical design specification and as a means to assure concurrence with standards. In addition, use of a checklist of common coding pitfalls, available from the SFI guidebook, helps reduce the chance of avoidable errors.

SOFTWARE TESTING AND ACCEPTANCE PHASE

Safety testing is performed to verify correct incorporation of software safety requirements and identify potential hazards not identified earlier in the lifecycle. Software safety testing is performed at the integration, Computer Software Component Item (CSCI) level, the system and acceptance levels. Testing at the system and acceptance level verifies correct operation of the SCCSCs in conjunction with system hardware and operators. It also verifies correct operational stress conditions in the presence of system faults. In addition, the testing verifies that safety related procedures incorporated in the user's manuals perform as expected.

Unacceptable hazards, as defined by the safety plan, identified during testing, are corrected and reverified prior to software delivery or usage.

SOFTWARE OPERATIONS AND MAINTENANCE

The above processes used to specify, develop, analyze and test SCCSCs are also used when changes are made to that software. The process includes updating the software safety requirements, identification of new SCCSCs, updating of specification, design, and operator documentation for SCCSCs, updating and adding comments for safety critical code, and testing of the SCCSCs. Testing includes regression testing to verify correct implementation of all software safety requirements.

SAFETY RELATED TASKS PERFORMED THROUGHOUT THE LIFECYCLE

A tracking system within the configuration management structure is used to ensure that software safety requirements are properly implemented and verified. A description of the implementation of each requirement is essential. The tracking of software hazard closures and verification should be accomplished through safety analysis documentation (e.g., hazard reports).

A project/program is required to conduct a series of formal safety reviews/audits to ensure that implementation of safety controls for hazards are adequate for the system and to approve any safety related waivers or deviations.

User interface analyses are performed to ensure that all necessary status related to safety functions is presented to the user to allow the user to take the necessary actions. Included is an evaluation of the User's Guide and Operational Procedures Manual to ensure 1) they contain any safety related operations to mitigate or respond to hazards, and 2) they do not contain procedures which initiate or contribute to hazards.

SUMMARY

While there remains much to do, NASA is taking the right steps to address software safety and the software process as a whole. The Software Safety Standard lays the foundation; next we need to work towards full compliance across NASA. While tailoring is permitted to account for project severity and size, a core safety activity and a minimum set of software safety requirements must remain. To meet NASA Administrator Daniel Goldin's challenge of "faster, better, cheaper, without compromising safety" we need to utilize software, with all its versatility, more and more heavily. Created within the right environment, with proper safety definition and analyses, methods like Software Formal Inspections, and standard software procedures and practices, NASA software can, and will be, safer and more reliable.

ACKNOWLEDGEMENTS

I wish to thank Mike Gilson and Tom Ziemianski for their patience as well as their editing and technical skills. Also, I wish to thank my management which has been very supportive.

REFERENCES

[1] NASA Safety Policy and Requirements Document, NHB 1700.1 (IV-B), June 1993.

[2] "An Assessment of the Space Shuttle Flight Software Development Processes" National Research Council Report. 1993.

[3] Lutz, Robyn R., "Analyzing Software Errors in Safety-Critical, Embedded Systems," Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA. February 1994.

[4] Software Formal Inspections Class Notes, FI-JPL/NASA, Rev A, 1992.

[5] Fagan, Michael E., "Design and Code Inspections to Reduce Errors in Program Development," from IBM Systems Journal, Vol. 15, No. 3, 1976, pp 182-211.

[6] Fagan, Michael E., "Advances in Software Inspections," IEEE Transactions on Software Engineering, Vol SE-12, No. 7, July 1986, pp. 744-751.

[7] Kelly, John C., Sherif, Joseph S., and Hops, Jonathan, "An Analysis of Defect Densities Found during Software Inspections," in J. Systems Software, Elservier Science Publishing Co., Inc, 1992: 17:111-117.

[8] Knisely, Merle H., Yang, Ling, and Chen H., Basilio, "Software Safety in Software Product Development," Hazard Prevention 1st Quarter 1994, pp. 20-27.

[9] Brown, Michael L., "Software System Safety Technical Review Panel," Hazard Prevention 1st Quarter 1994, pp. 8-11.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1994	Technical Memorandum

4. TITLE AND SUBTITLE

Implementing Software Safety in the NASA Environment

5. FUNDING NUMBERS

WU-323-88-03

6. AUTHOR(S)

Martha S. Wetherholt and Charles F. Radley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E-8870

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM-106597

11. SUPPLEMENTARY NOTES

Prepared for the Safety Through Quality Conference sponsored by the Real Time Associates, Ltd., Old Windsor, Berkshire, United Kingdom, June 6-7, 1994. Martha S. Wetherholt, NASA Lewis Research Center and Charles F. Radley, EBASCO Services, Inc., 2001 Aerospace Parkway, Brook Park, Ohio 44142 (work funded by NASA Contract NAS3-26764). Responsible person, Martha S. Wetherholt, organization code 0152, (216) 433-2416.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Categories 38 and 66

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Until recently, NASA did not consider allowing computers total control of flight systems. Human operators, via hardware, have constituted the ultimate safety control. In an attempt to reduce costs, NASA has come to rely more and more heavily on computers and software to control space missions. (For example, software is now planned to control most of the operational functions of the International Space Station.) Thus the need for systematic software safety programs has become crucial for mission success. Concurrent engineering principles dictate that safety should be designed into software up front, not tested into the software after the fact. 'Cost of Quality' studies have statistics and metrics to prove the value of building quality and safety into the development cycle. Unfortunately, most software engineers are not familiar with designing for safety, and most safety engineers are not software experts. Software written to specifications which have not been safety analyzed is a major source of computer related accidents. Safer software is achieved step by step throughout the system and software lifecycle. It is a process that includes requirements definition, hazard analyses, formal software inspections, safety analyses, testing, and maintenance. The greatest emphasis is placed on clearly and completely defining system and software requirements, including safety and reliability requirements. Unfortunately, development and review of requirements are the weakest link in the process. While some of the more academic methods, e.g. mathematical models, may help bring about safer software, this paper proposes the use of currently approved software methodologies, and sound software and assurance practices to show how, to a large degree, safety can be designed into software from the start. NASA's approach today is to first conduct a preliminary system hazard analysis (PHA) during the concept and planning phase of a project. This determines the overall hazard potential of the system to be built. Shortly thereafter, as the system requirements are being defined, the second iteration of hazard analyses takes place, the systems hazard analysis (SHA). During the systems requirements phase, decisions are made as to what functions of the system will be the responsibility of software. This is the most critical time to affect the safety of the software. From this point, software safety analyses as well as software engineering practices are the main focus for assuring safe software. While many of the steps proposed in this paper seem like just sound engineering practices, they are the best technical and most cost effective means to assure safe software within a safe system.

14. SUBJECT TERMS

Safety; NASA; Lifecycle; Software

15. NUMBER OF PAGES

19

16. PRICE CODE

A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	